0400
06-01-01.

500.40122X00

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s): I. KYUSHIMA, ET AL.

Serial No.: 09 / 854,458

Filed: MAY 15, 2001

Title: "COMPILE METHOD SUITABLE FOR SPECULATION MECHANISM".

LETTER CLAIMING RIGHT OF PRIORITY

Assistant Commissioner for
Patents
Washington, D.C. 20231

MAY 30, 2001

Sir:

Under the provisions of 35 USC 119 and 37 CFR 1.55, the applicant(s) hereby claim(s) the right of priority based on:

Japanese Patent Application No. 2000 - 148588
Filed: MAY 16, 2000

A certified copy of said Japanese Patent Application is attached.

Respectfully submitted,

ANTONELLI, TERRY, STOUT & KRAUS, LLP

Carl I. Brundidge
Registration No. 29,621

CIB/rp
Attachment

E5926-01 ES

日　本　国　特　許　庁
JAPAN PATENT OFFICE

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed with this Office

出願年月日
Date of Application: ２０００年　５月１６日

出願番号
Application Number: 特願２０００－１４８５８８

出願人
Applicant(s): 株式会社日立製作所

２００１年　４月２０日

特許庁長官
Commissioner,
Japan Patent Office 及川耕造

出証番号　出証特２００１－３０３２９５５

【書類名】　特許願
【整理番号】　K00000711
【提出日】　平成12年 5月16日
【あて先】　特許庁長官殿
【国際特許分類】　G06F　9/45
【請求項の数】　7
【発明者】
　【住所又は居所】　神奈川県川崎市麻生区王禅寺１０９９番地　株式会社日立製作所　システム開発研究所内
　【氏名】　久島　伊知郎
【発明者】
　【住所又は居所】　神奈川県川崎市麻生区王禅寺１０９９番地　株式会社日立製作所　システム開発研究所内
　【氏名】　西山　博泰
【特許出願人】
　【識別番号】　000005108
　【氏名又は名称】　株式会社日立製作所
【代理人】
　【識別番号】　100075096
　【弁理士】
　【氏名又は名称】　作田　康夫
【手数料の表示】
　【予納台帳番号】　013088
　【納付金額】　21,000円
【提出物件の目録】
　【物件名】　明細書　1
　【物件名】　図面　1
　【物件名】　要約書　1
【プルーフの要否】　要

【書類名】　明細書

【発明の名称】　投機機構向けコンパイル方法

【特許請求の範囲】

【請求項１】　投機命令および投機誤りをチェックするチェック命令を有するプロセッサ向けのコードを生成するコンパイラにおいて、

（１）プログラム中の繰り返し実行される部分に対し、投機命令およびチェック命令を使用したコード(a)と、投機命令およびチェック命令を使用しないコード(b)を含む、少なくとも２種類のパターンのコードを生成する処理と、

（２）コード(a)の実行中に投機誤りチェックにかかった回数が特定の条件を満たす場合に、以降の繰り返しではコード(b)を実行するように、制御の移行を行うコードを生成する処理

を含むことを特徴とする、投機機構向けコンパイル方法。

【請求項２】　請求項１のコンパイル方法において、（２）における特定の条件として、投機チェックにかかった回数が一定値を超えることを条件とする、投機機構向けコンパイル方法。

【請求項３】　請求項１のコンパイル方法において、（２）における特定の条件として、繰り返し部分の実行回数に対する、投機チェックにかかった回数の割合が一定値を超えることを条件とする、投機機構向けコンパイル方法。

【請求項４】　請求項１のコンパイル方法において、投機チェックにかかった場合、カウンタを更新し、その値が一定値を越えた場合には、コード(b)に制御を移すことを特徴とする、投機機構向けコンパイル方法。

【請求項５】　請求項１のコンパイル方法において、投機チェックにかかった場合、ただちにコード(b)に制御を移すことを特徴とする、投機機構向けコンパイル方法。

【請求項６】　請求項１のコンパイル方法を用いたコンパイラ。

【請求項７】　請求項６のコンパイラを格納した記憶媒体。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】

本発明は、計算機の利用技術において、オブジェクトプログラムの実行時間を削減するコンパイル方法に関する。特に、投機機構を持つ計算機向けのコンパイル方法に関する。

【０００２】

【従来の技術】

近年のマイクロプロセッサは、オブジェクトプログラムを高速に実行することを目的として、特定の命令（主にロード命令）を投機的に実行する命令（投機ロード命令）と、その投機実行が誤りであったかどうかをチェックする命令（チェック命令）を備えているものがある。これらを合わせて投機機構と呼ぶ。投機機構およびそれらを用いた最適化方法については、たとえばIntel: IA－64 Application Developers Architecture Guide, May 1999, Order Number: 245188－001の 10－4節および10－5節に記載がある。

【０００３】

投機機構を用いた最適化の一例として、不明なデータ依存関係がある場合のループ不変式移動がある。これを図２のプログラム片を用いて示す。図２のプログラムはC言語で記載されており、(201)の条件condが満たされている間、(202)～(204)の文を繰り返し実行することを意味するループである。(202)では、a+bの値を計算し、cに代入する。(203)では、cの値を、pが表すアドレスで示されるメモリ（*p）に書きこむ。(204)では、pの値を更新する。

【０００４】

図１のプログラム片に対して、投機機構を利用しないでオブジェクトコードを生成すると、図３のようになる。なお図３ではコードをわかりやすくするために一部C言語の構文を用いて記述している。図３では(301)の条件condが満たされている間、(302)～(306)の命令を繰り返し実行する。(302)では、変数aのアドレス（&a）で示されるメモリ内容、即ちaの値をレジスタr1へロードする。(303)では、同様にbの値をメモリからレジスタr2へロードする。(304)ではr1とr2の和を計算し、r3へ代入する。(305)では、r3の値をpが表すアドレスで示されるメモリにストアする。(306)では、pの値を更新する。

【０００５】

図３のコードでは、は、ループの繰り返しで毎回同じ値を計算している、即ちループ不変式である可能性が高いので、これらの命令をループ外に移動する（ループに入る前に1度だけ計算をしておき、ループの中では計算された値を使う）ようにすれば、実行時間が短縮される。しかし、pが表すアドレスと、aまたはbのアドレスが一致している場合には、このような移動はできない。すなわち、(305)のストア先のメモリのアドレスと、(302)または(303)でロードするメモリのアドレスが一致していた場合、*pへの書き込みでaまたはbの値が書きかえられるので、(302)から(304)で計算される値はループ不変でなくなり、(302)から(304)の命令をループ外への移動することはできない。このように、不明なデータ依存関係がある（ストア先のメモリのアドレスと、ロードするメモリのアドレスが一致するかどうかがわからない）場合、一般にコンパイラはループ不変式移動を行うことはできない。

【０００６】

一方、投機機構を利用した場合、図１のプログラム片に対してループ不変式移動を行うことができる。これを図４に示す。図４では、aおよびbのロードと加算が、ループ外に移動されている（(401)～(403)）。そのときのロード命令は通常のロード命令ではなく、投機的なものであるのでld.a命令を用いている（aはadvancedを意味する）。またループ内では(405)と(406)命令で投機ロードが不正でないかどうかをチェックするチェック命令（(405),(406)）が置かれている。(405)のchk.a r1, recover1は、r1に対するld.a命令から現在のchk.a命令までの間に、la.dでロードしたメモリアドレスに対するストアがあったかどうかをチェックする。あった場合には、投機実行は誤りであったので、recover1（(410)）に分岐する。分岐先のrecover1では、(411)でaの値をロードし直し、(412)でa+bを計算し直し、(413)でチェック命令の次の命令(406)に戻る。(407)および(414)～(417)も同様である。なお、(410)～(413)および(414)～(417)は、投機チェック命令で引っかかった（投機実行が誤りであった）場合に実行をやり直すので、回復コードと呼ばれる。

【０００７】

図４のコードでは、投機チェックにひっかかって回復コードに分岐しない限り、ループ中でロード命令や加算命令が実行されることがないので、図３のコードに比べて実行時間が短くなることが期待される（ループ中の命令は図３に比べて1命令しか減少していないが、一般にチェック命令はロード命令や加算命令に比べて少ないサイクル数で行えるので、命令数以上に効果があると期待される）。

このように、投機機構を用いれば、ロード命令とストア命令の間に不明な依存関係がある場合にも、ループ不変式移動を行える。また、ループ不変式移動以外にも、ロード命令とストア命令の間に不明な依存関係がある場合に、ロード命令を、ストア命令を越えて移動するなどの命令スケジューリングが行えるようになる。

【０００８】

【発明が解決しようとする課題】

従来技術ではしかしながら、投機チェック命令にひっかかって回復コードに分岐することが頻繁に行われると、却って実行速度が低下する可能性があるという問題がある。たとえば図４のコードでは、(405)または(406)のchk.a命令で、回復コードに分岐することが頻繁に行われると、分岐によるオーバーヘッドや、再計算のため、性能が却って低下する可能性がある。

【０００９】

そこで本発明の目的は、投機機構を利用したコードを生成するコンパイラにおいて、図２のように繰り返し実行される部分について、頻繁に投機チェックにひっかかり回復コードに分岐することで性能が悪くなる、ということが生じないコードを生成するコンパイル方法を提供することである。

【００１０】

【課題を解決するための手段】

前記目的を達成するため、本発明のコンパイル方法では、以下を行う。

【００１１】

（１）コンパイラは、ループのように繰り返し実行される部分に対し、投機機構（投機命令および投機チェック命令）を利用したコード(a)と、投機機構を利

用しないコードの２種類のパターンのコード(b)を生成する。最初は投機機構を利用したコード(a)を実行するようにする。

【００１２】

（２）投機機構を利用したコード(a)中の投機チェック命令にひっかかった場合に実行される回復コードでは、チェックにひっかかった回数をカウントし、回数が上限値を越えている場合は、以降は投機機構を利用しないパターンのコード(b)を実行するようにする。

【００１３】

これにより、投機チェックにひっかかる回数がある値を超えた場合は、以降は投機機構を利用しないコードが実行されるので、頻繁に投機チェックにひっかかり回復コードに分岐することで性能が悪くなることがなくなる。また投機チェックにひっかかる回数が少ない場合には、投機機構を利用したコードが実行されるので、投機機構を利用しない場合より実行速度が速くなる。なお回数の上限値を１にすれば、回数をカウントする必要はなくなる。また回数ではなく、チェックにひっかかった確率を用いてもよい。

【００１４】

【発明の実施の形態】

以下、本発明の一実施例として、投機機構を利用したループ不変式移動を行うコンパイラを説明する。

【００１５】

図１は、本発明によるコンパイラが稼動する計算機システムの構成図である。図示するように、計算機システムはＣＰＵ１０１、主記憶装置１０４、外部記憶装置１０５、ディスプレイ装置１０２、キーボード１０３より構成されている。外部記憶装置１０５にはソースプログラム１０６、オブジェクトプログラム１０７が格納される。主記憶装置１０４には、コンパイラ１０８と、コンパイル処理過程で必要となる中間コード１０９が保持される。コンパイル処理はＣＰＵ１０１がコンパイラプログラム１０８を実行することにより行われる。キーボード１０３はユーザからのコマンドをコンパイラ１０８に与えるのに用いる。ディスプレイ装置１０２はコンパイルの終了またはエラーをユーザに知らせる。

【００１６】

図５は、コンパイル処理の流れを示したフローチャートである。コンパイラの処理は、まずステップ５０１で構文解析を行う。構文解析はソースプログラム１０６を読み出し、コンパイラ内部で処理可能な中間コード１０９を作成する。構文解析処理については、たとえば「エイホ、セシィ、ウルマン著：コンパイラＩ（サイエンス社、１９９０年）３０頁～７４頁」に記載されているので、ここでは詳しく説明しない。次にステップ５０２で、ループ解析を行う。ループ解析処理についても「エイホ、セシィ、ウルマン著：コンパイラII（サイエンス社、１９９０年）７３４頁～７３７頁」に記載があるのでここでは詳しく説明しない。ループ解析により、プログラムに含まれるループの集合が求められる。次にステップ５０３で、未処理のループがあるか調べる。なければステップ５０６へ進み、オブジェクトコードを生成して終了する。オブジェクトコード生成については、同じく「エイホ、セシィ、ウルマン著：コンパイラII（サイエンス社、１９９０年）６２４頁～７０７頁」に記載があるので、ここでは詳しく説明しない。未処理のループがあればステップ５０４でループを１つ取り出す。そしてステップ５０５で、投機機構を利用したループ不変式移動処理を行う。ステップ５０５の処理については図７を用いて詳しく説明する。この処理の後はステップ５０３から繰り返す。

【００１７】

図６は本実施例におけるコンパイラの中間コードの例である。中間コードは構文解析５０１の処理により作成される。図６の中間コードは図２のソースプログラムに対応している。図６の中間コードは、基本ブロック（Ｂａｓｉｃ Ｂｌｏｃｋ、ＢＢと略される）をエッジで結んだグラフで表現されている。（このようなグラフは制御フローグラフと呼ばれている。）６０１から６０４は基本ブロックである。これらの基本ブロックには、ＢＢ１からＢＢ４までの番号がそれぞれ付けられている。基本ブロックは途中で分岐や飛び込みのない、一連のコード列を表している。エッジ（矢印）は基本ブロック間の遷移を表している。たとえば基本ブロック６０１から６０２にエッジが張られているので、６０１が終った後で、６０２へ制御が移ることを示している。基本ブロックの解析方法や制御フロ

ーグラフの構成方法については前著（コンパイラII）６４２頁～６４８頁に記載されているので、ここでは詳しく述べない。各基本ブロック中に書かれているものは実行文であり、その基本ブロックに制御が移ったときに実行される。各文の左側（S1～S7）は文番号を表す。

【００１８】

図７は投機機構を利用したループ不変式移動処理５０５の流れを詳しく示したフローチャートである。まずステップ７０１で、ループ中に未処理の文（命令）があるか調べる。なければ終了する。あればステップ７０２に進み、未処理の文を１つ取り出し、ステップ７０３で、取り出した文がループ不変式であるか調べる。ループ不変式であるかどうかは、すべてのオペランドがループ不変であるかを調べる。ロード命　令の場合はロードするメモリアドレスがループ不変であるかを調べる。ただしアドレスがループ不変であっても、明らかなデータ依存（同じアドレスへのストア）がループ中にあればループ不変でないとする。（不明なデータ依存のある場合はループ不変式とみなす。）ループ不変でなければステップ７０１から繰り返す。ループ不変であれば、それがロード命令でかつ、不明なデータ依存がある（ロードするメモリアドレスへのストアがループ中で実行される可能性がある）か調べる、そうであればステップ７０５に進む。ステップ７０５では、すでにループの２重化（ループの複写）を行っているか調べる。２重化を行っていない場合はステップ７０６に進み、ループの２重化を行う。これは、元のループの後に、複写ループを作るもので、たとえば図６の中間コードでは、ステップ７０６を行った後の中間コードは図８のようになる。図８ではBB5（804）およびBB8（805）が複写されたループを構成する。さらに元のループの前に、カウンタをゼロクリアするコード（807のS15）を挿入する。次にステップ７０７で、移動対象のロード命令をループ外に移動する。その際、ロード命令を投機ロード命令（load.a）に変更する。次にステップ７０８で、元のロード命令があった位置にチェック命令（chk.a）を置くとともに、チェックにひっかかった場合の回復コードを作成する。これを図９に示す。

【００１９】

図９では、チェック命令（904のS16）から回復コード（906）への分岐が作成

されている。回復コードの先頭では、カウンタをインクリメントし（906のS17）、カウンタ値が一定値を超えた場合には複写ループへ分岐するコード（906のS18）が作成されている。カウンタが一定値を超えていない場合にはaの再ロード（907のS19）を行い、チェック命令の次の命令（905のS3）に戻る。

【００２０】

図７の説明に戻る。ステップ７０５でループの２重化をすでに行っている場合は、ステップ７０７から処理を行う。図８の中間語の例では、最初のロード命令（S2）の移動ではループの複写を行うが、２番目のロード命令（S3）の移動ではループ２重化処理はスキップされる。ステップ７０９では、従来のループ不変式移動と同様に、移動対象文をループ外に移動する。さらに、ステップ７１０で、移動対象文で参照されるオペランドが、回復コード中で設定されているかどうかを調べ、設定されている場合は、ステップ７１１で、回復コードの該設定文の直後にも、命令をコピーする。たとえば、905のS4の文（t3=t1+t2）をステップ７０９でループ外に移動する場合、そのオペランドのt1，t2は回復コードのS19（t1=load.a(&a)）で中で設定されているので、その直後にもコピーされる。図７の処理を行うことにより、図８の中間コードは最終的に図１０のようになる。

【００２１】

図１１は、図１０の中間語をオブジェクトコードにしたもので、本発明のコンパイラにより生成されるものである。ここでは図２，３と同様、コードをわかりやすくするために一部C言語の構文を用いて記述している。図１１のプログラムでは、投機機構を利用してループ不変式移動がされたループ（1105～1110）と、投機機構を利用しないループ（1112～1118）の２つのループがあり、最初は投機機構を利用したループが実行される。投機機構を利用したループ内の最初のチェック命令（1106）にひっかかったときに分岐する回復コード（1120～1125）では、最初にカウンタ値を更新し（1121）、それが一定値を超えた場合は、投機機構を利用しないループに制御を移す（1122）。２つめのチェック命令（1107）についても同様である。これにより、頻繁にチェックにひっかかる場合には投機機構を利用しないループが実行されるので、以降は、投機チェックからの回復のために実行速度が低下することがない。また投機チェックにひっかかる回数が少ない

場合には、投機機構を利用したコード（ループ不変式移動がされている）が実行されるので、投機機構を利用しない場合より実行速度が速くなる。

【００２２】

以上の実施例では、カウンタを用いて投機チェックにひっかかった回数をカウントしていたが、投機チェックにひっかかる回数の上限値を１にした場合は、カウンタの更新処理は不要になる。すなわち、１度でもチェックにひっかかった場合、直に投機コードを利用しないコードに分岐すればよい。すなわち、ステップ７０８では回復コードを生成せずに、チェック命令から複写ループへ直接分岐するようにする。この場合に生成されるオブジェクトコードは図１２に示すようになる。図１２のプログラムでは、投機機構を利用してループ不変式移動がされたループ（1204～1209）と、投機機構を利用しないループ（1211～1217）の２つのループがあり、投機機構を利用したループ内のチェック命令（1205、1206）にひっかかったときには、投機機構を利用しないループに直接分岐する。このようにすると、回復コードでのカウンタ更新やカウンタ値の比較コードが不要になるという利点がある。

【００２３】

また以上の実施例では、投機チェックにひっかかった回数を閾値にしていたが、回数ではなく、確率（割合）を閾値にしてもよい。すなわち、ループの実行回数をN、チェックにひっかかった回数をMとし、M/Nが一定値以上になっているかを調べる。この場合のオブジェクトコードは図１３に示すようになる。図１３のプログラムでは、投機機構を利用してループ不変式移動がされたループ（1306～1312）と、投機機構を利用しないループ（1315～1321）の２つのループがあり、投機機構を利用したループでは、ループの実行回数をカウントする（1307）。投機機構を利用したループ内の最初のチェック命令（1308）にひっかかったときに分岐する回復コード（1323～1329）では、チェックにひっかかった回数を表すカウンタ値を更新し（1324）、それをループ実行回数で割り（1325）、割った値が一定値を超えた場合は、投機機構を利用しないループに戻る（1326）。２つめのチェック命令（1309）も同様である。このようにした場合、ループ実行回数をカウントするオーバーヘッドや除算のオーバーヘッドが加わるものの、チェックに

ひっかかる確率で判断ができるので、どちらのループを実行すべきかをより精密に判断することが可能になるという利点がある。

【００２４】

以上の実施例は、投機機構を利用してループ不変式移動を行う場合に本発明を適用したものであったが、本発明はこれに限定されるものではなく、投機機構を利用して、ループ内で命令を移動する（命令スケジューリング）場合にも適用できる。命令スケジューリングは、命令の順序を並べ替えることにより、命令レイテンシを隠蔽し、命令列の実行時間を削減する最適化である。一般に、ストア命令と、後続するロード命令の間に不明な依存関係がある場合（それぞれの命令で参照するメモリアドレスが一致している可能性がある場合）は、ロード命令をストア命令の前に移動することはできないが、投機機構を利用すれば命令の順序を入れ替えることができる。すなわち、ストア命令の前に投機的ロード命令を実行し、ストア命令の後でチェック命令を実行すればよい。ループに対してこのような投機機構を利用した命令スケジューリングに本発明を適用した場合、投機機構を利用したループと、投機機構を利用しないループの２つのループコードを生成し、投機機構を利用したループ内でのチェック命令にひっかかった回数がある条件を満たす場合は、以降は投機機構を利用しないループに分岐するようにする。これにより、投機チェックに頻繁にひっかかる場合の性能低下を防ぐことができる。

【００２５】

【発明の効果】

本発明のコンパイル方法によれば、ループのように繰り返し実行される部分に対して投機機構を利用したコードを生成する場合、頻繁に投機チェックにひっかかり回復コードに分岐するために性能が低下する、ということが発生しないコードを生成できる。

【図面の簡単な説明】

【図１】

本発明のコンパイラが稼動する計算機システムの構成図。

【図２】

ソースプログラム例を示す図。

【図３】

投機機構を利用しない場合の従来技術の生成コードを示す図。

【図４】

投機機構を利用した場合の従来技術の生成コードを示す図。

【図５】

コンパイル処理の流れを示す図。

【図６】

図２のプログラムの対する中間コードの例を示す図。

【図７】

本発明を適用したループ不変式移動処理の流れを示す図。

【図８】

ループ２重化直後の中間コードの例を示す図。

【図９】

最初のロード命令をループ外移動した直後の中間語の例を示す図。

【図１０】

ループ不変式移動処理後の中間コードの例を示す図。

【図１１】

本発明を適用した場合の生成コードの例を示す図。

【図１２】

本発明を適用した場合の別の生成コードの例を示す図。

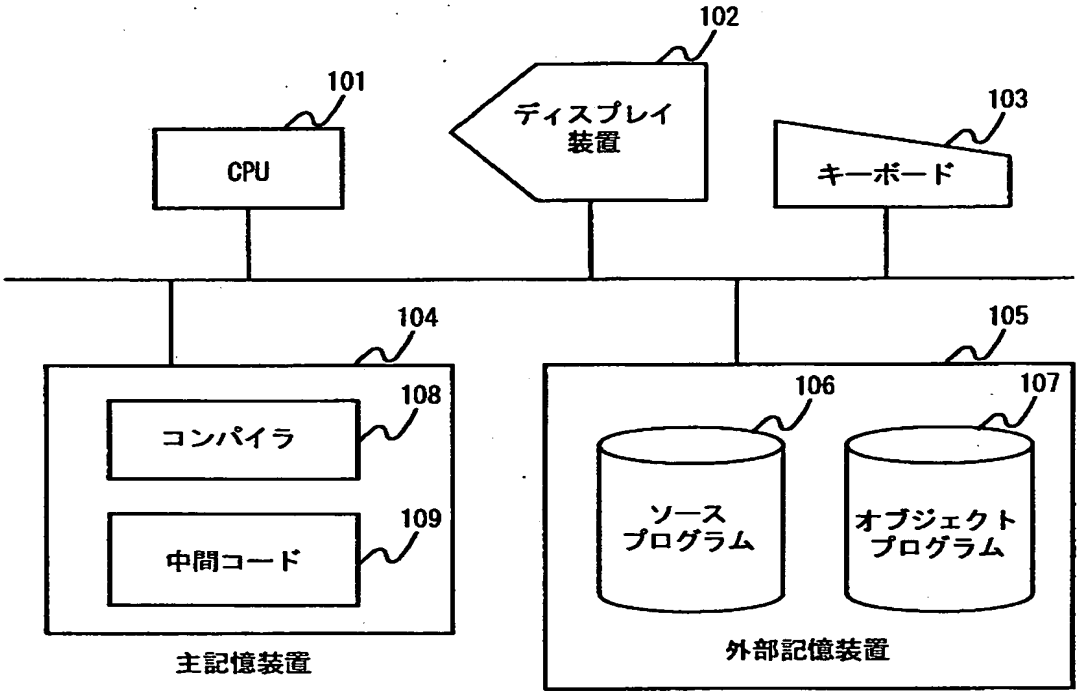
【図１３】

本発明を適用した場合の別の生成コードの例を示す図。

【書類名】　図面

【図１】

図　1



【図２】

図　2

```
while (cond) {                    (201)
    c = a+b;                      (202)
    *p = c;                       (203)
    p = p->next;                  (204)
}                                 (205)
```

【図３】

図 3

```
while (cond) {                (301)
   ld    r1=[&a]              (302)
   ld    r2=[&b]              (303)
   add   r3=r1,r2             (304)
   st    *p=r3                (305)
   p = p->next                (306)
}                             (307)
```

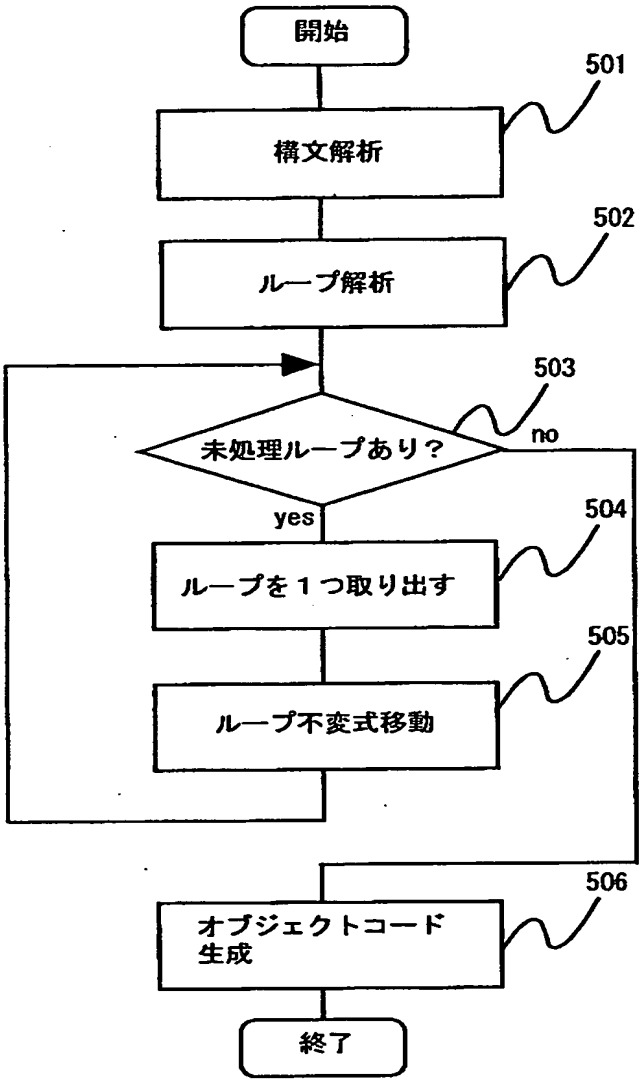
【図４】

図　4

```
ld.a   r1=[&a]                    (401)
ld.a   r2=[&b]                    (402)
add    r3=r1,r2                   (403)
while (cond) {                    (404)
    chk.a   r1,recover1           (405)
L1: chk.a   r2,recover2           (406)
L2: st      *p=r3                 (407)
    p = p->next                   (408)
}                                 (409)
-------------------------------------
recover1:                         (410)
ld.a   r1=[&a]                    (411)
add    r3=r1,r2                   (412)
br     L1                         (413)
-------------------------------------
recover2:                         (414)
ld.a   r2=[&b]                    (415)
add    r3=r1,r2                   (416)
br     L2                         (417)
```

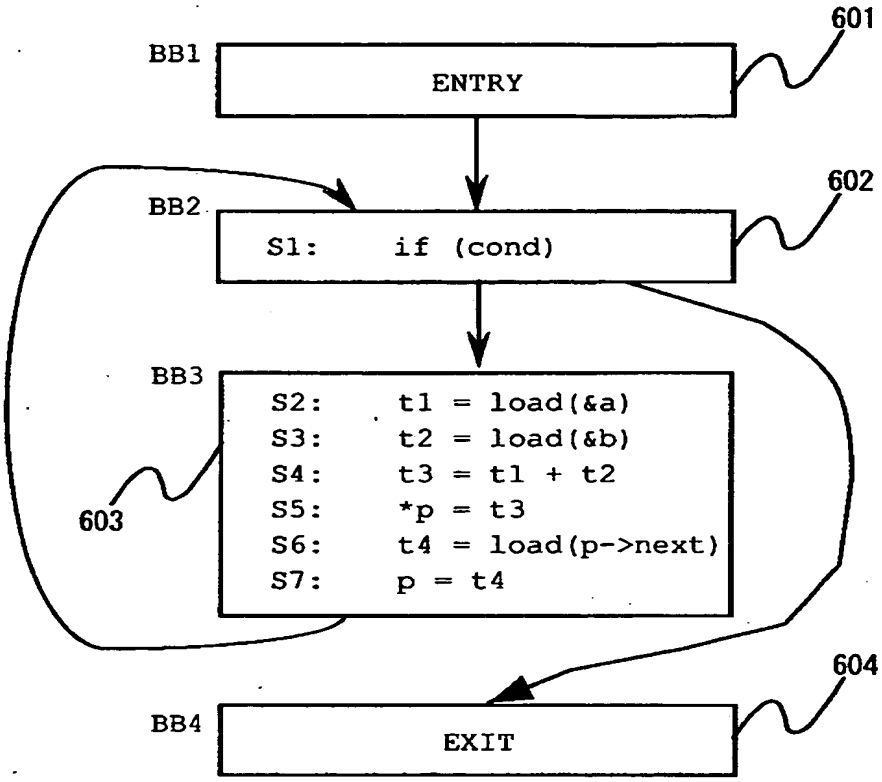
【図５】

図　５


開始
501
構文解析
502
ループ解析
503
未処理ループあり？
no
yes
504
ループを１つ取り出す
505
ループ不変式移動
506
オブジェクトコード
生成
終了

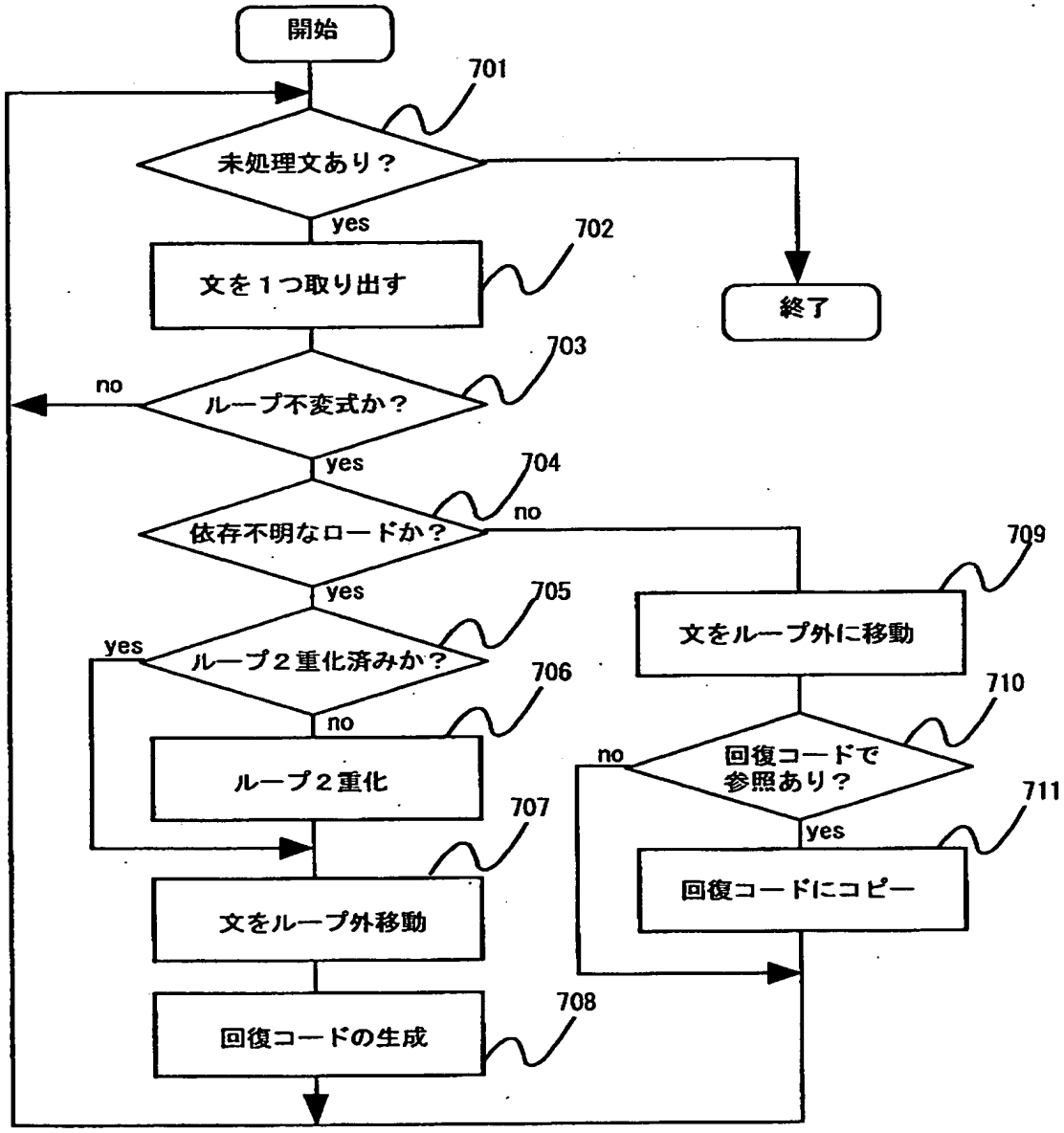
【図６】

図 6


601
BB1
ENTRY
602
BB2
S1: if (cond)
BB3
S2: t1 = load(&a)
S3: t2 = load(&b)
S4: t3 = t1 + t2
S5: *p = t3
S6: t4 = load(p->next)
S7: p = t4
603
604
BB4
EXIT

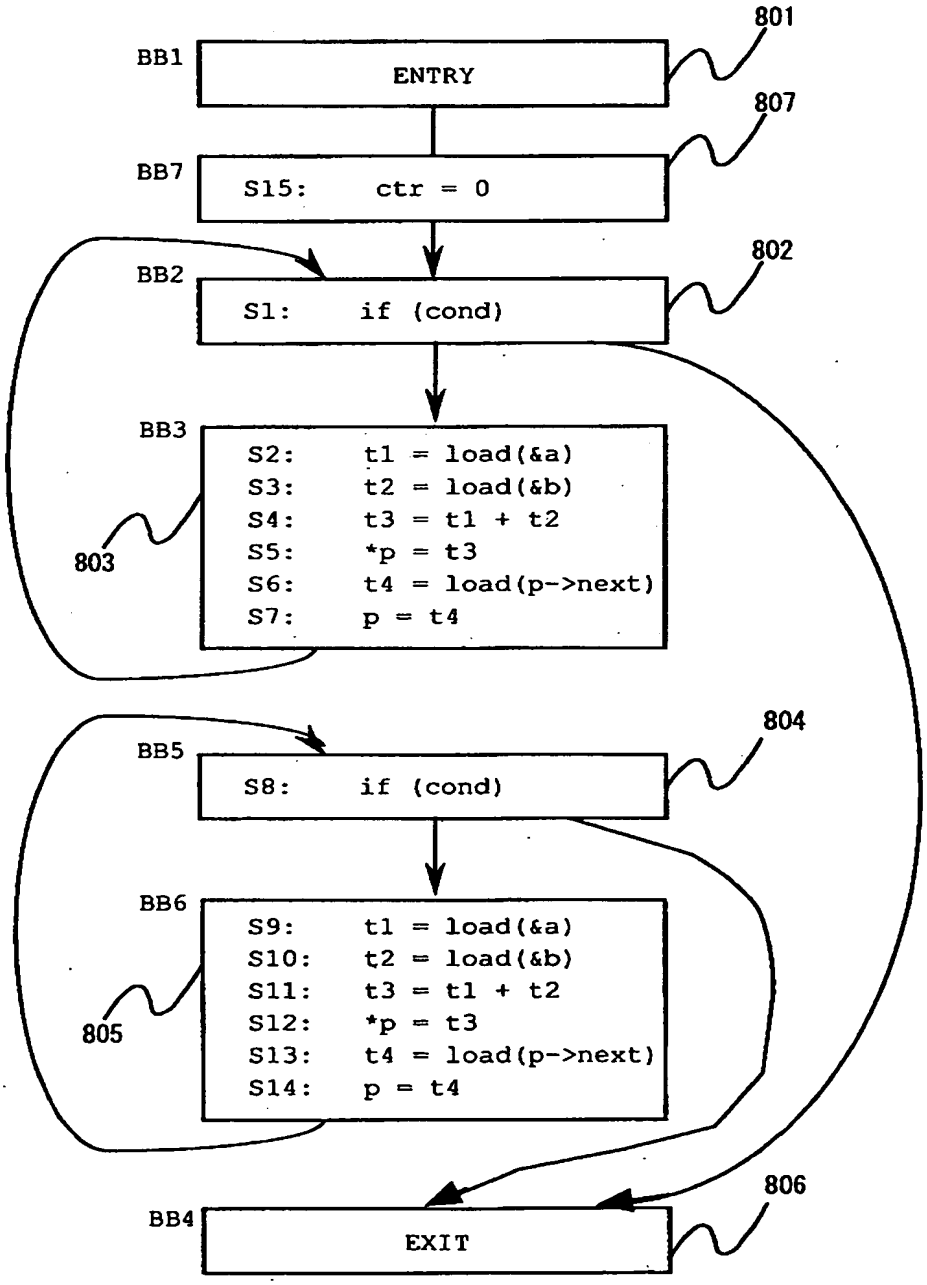
【図７】

図　７


開始
701
未処理文あり？
yes
no
702
文を１つ取り出す
終了
703
ループ不変式か？
no
yes
704
依存不明なロードか？
no
yes
709
文をループ外に移動
705
ループ２重化済みか？
yes
706
no
ループ２重化
710
回復コードで
参照あり？
no
711
yes
回復コードにコピー
707
文をループ外移動
708
回復コードの生成

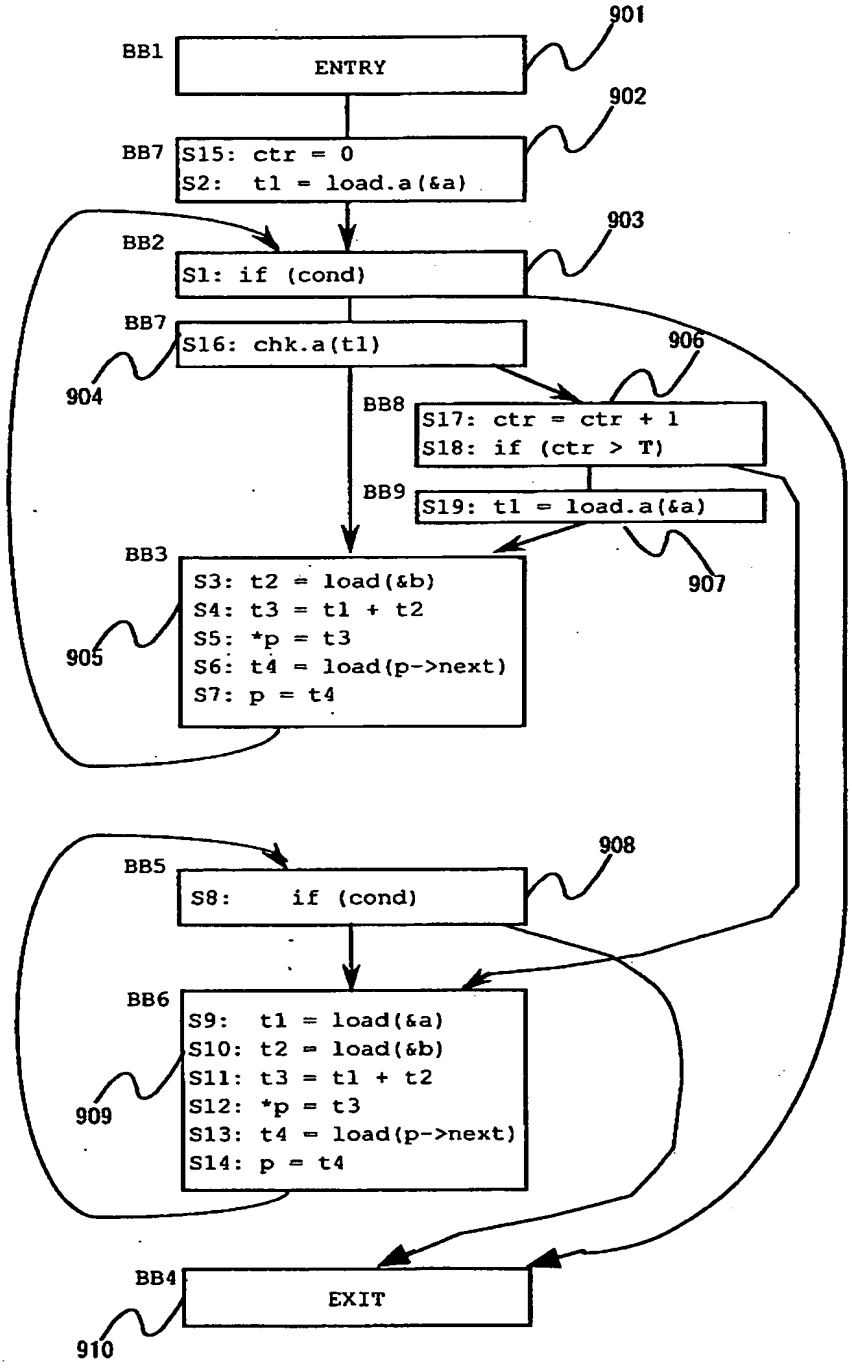
【図８】

図　８


BB1
ENTRY
801
807
BB7
S15: ctr = 0
BB2
S1: if (cond)
802
BB3
S2: t1 = load(&a)
S3: t2 = load(&b)
S4: t3 = t1 + t2
S5: *p = t3
S6: t4 = load(p->next)
S7: p = t4
803
BB5
S8: if (cond)
804
BB6
S9: t1 = load(&a)
S10: t2 = load(&b)
S11: t3 = t1 + t2
S12: *p = t3
S13: t4 = load(p->next)
S14: p = t4
805
BB4
EXIT
806

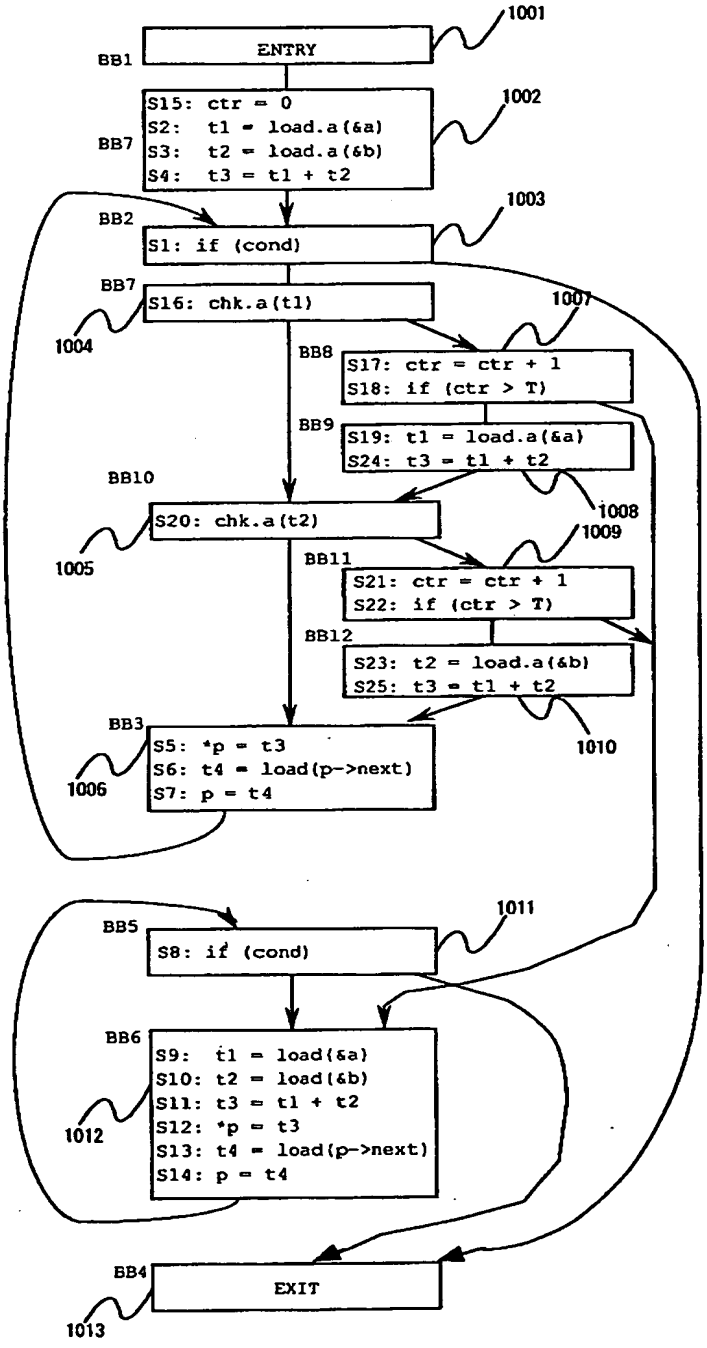
【図９】

図　９


BB1
ENTRY
901
BB7
S15: ctr = 0
S2: t1 = load.a(&a)
902
BB2
S1: if (cond)
903
BB7
S16: chk.a(t1)
904
906
BB8
S17: ctr = ctr + 1
S18: if (ctr > T)
BB9
S19: t1 = load.a(&a)
907
BB3
S3: t2 = load(&b)
S4: t3 = t1 + t2
S5: *p = t3
S6: t4 = load(p->next)
S7: p = t4
905
908
BB5
S8: if (cond)
BB6
S9: t1 = load(&a)
S10: t2 = load(&b)
S11: t3 = t1 + t2
S12: *p = t3
S13: t4 = load(p->next)
S14: p = t4
909
BB4
EXIT
910

【図１０】

図 10


BB1
ENTRY
1001
BB7
S15: ctr = 0
S2: t1 = load.a(&a)
S3: t2 = load.a(&b)
S4: t3 = t1 + t2
1002
BB2
S1: if (cond)
1003
BB7
S16: chk.a(t1)
1004
1007
BB8
S17: ctr = ctr + 1
S18: if (ctr > T)
BB9
S19: t1 = load.a(&a)
S24: t3 = t1 + t2
1008
BB10
S20: chk.a(t2)
1005
1009
BB11
S21: ctr = ctr + 1
S22: if (ctr > T)
BB12
S23: t2 = load.a(&b)
S25: t3 = t1 + t2
1010
BB3
S5: *p = t3
S6: t4 = load(p->next)
S7: p = t4
1006
BB5
S8: if (cond)
1011
BB6
S9: t1 = load(&a)
S10: t2 = load(&b)
S11: t3 = t1 + t2
S12: *p = t3
S13: t4 = load(p->next)
S14: p = t4
1012
BB4
EXIT
1013

【図１１】

図　11

```
copy  r4=0                      (1101)
ld.a  r1=[&a]                   (1102)
ld.a  r2=[&b]                   (1103)
add   r3=r1,r2                  (1104)
while (cond) {                  (1105)
    chk.a  r1,recover1          (1106)
L1: chk.a  r2,recover2          (1107)
L2: st     *p=r3                (1108)
    p = p->next                 (1109)
}                               (1110)
goto exit                       (1111)

while (cond) {                  (1112)
L3: ld   r1=[&a]                (1113)
    ld   r2=[&b]                (1114)
    add  r3=r1,r2               (1115)
    st   *p=r3                  (1116)
    p = p->next                 (1117)
}                               (1118)
exit:                           (1119)
-------------------------------------
recover1:                       (1120)
add   r4=r4,1                   (1121)
bc    r4>T,L3                   (1122)
ld.a  r1=[&a]                   (1123)
add   r3=r1,r2                  (1124)
br    L1                        (1125)
-------------------------------------
recover2:                       (1126)
add   r4=r4,1                   (1127)
bc    r4>T,L3                   (1128)
ld.a  r2=[&b]                   (1129)
add   r3=r1,r2                  (1130)
br    L2                        (1131)
```

【図１２】

図　12

```
ld.a   r1=[&a]                (1201)
ld.a   r2=[&b]                (1202)
add    r3=r1,r2               (1203)
while (cond) {                (1204)
    chk.a   r1,L3             (1205)
    chk.a   r2,L3             (1206)
    st      *p=r3             (1207)
    p = p->next               (1208)
}                             (1209)
goto exit                     (1210)

while (cond) {                (1211)
L3: ld    r1=[&a]             (1212)
    ld    r2=[&b]             (1213)
    add   r3=r1,r2            (1214)
    st     *p=r3              (1215)
    p = p->next               (1216)
}                             (1217)
exit:                         (1218)
```

【図１３】

図 13

```
copy  r4=0                    (1301)
copy  r5=0                    (1302)
ld.a  r1=[&a]                 (1303)
ld.a  r2=[&b]                 (1304)
add   r3=r1,r2                (1305)
while (cond) {                (1306)
    add     r5=r5,1           (1307)
    chk.a   r1,recover1       (1308)
L1: chk.a   r2,recover2       (1309)
L2: st      *p=r3             (1310)
    p = p->next               (1311)
}                             (1312)
goto exit                     (1314)

while (cond) {                (1315)
L3: ld    r1=[&a]             (1316)
    ld    r2=[&b]             (1317)
    add   r3=r1,r2            (1318)
    st     *p=r3              (1319)
    p = p->next               (1320)
}                             (1321)
exit:                         (1322)
-----------------------------
recover1:                     (1323)
add   r4=r4,1                 (1324)
div   r6=r4,r5                (1325)
bc    r6>T,L3                 (1326)
ld.a  r1=[&a]                 (1327)
add   r3=r1,r2                (1328)
br    L1                      (1329)
-----------------------------
recover2:                     (1330)
add   r4=r4,1                 (1331)
div   r6=r4,r5                (1332)
bc    r6>T,L3                 (1333)
ld.a  r2=[&b]                 (1334)
add   r3=r1,r2                (1335)
br    L2                      (1336)
```

【書類名】　要約書

【要約】

【課題】ループのように繰り返し実行される部分に対して投機機構を利用したコードを生成する場合に、頻繁に投機チェックにひっかかり回復コードに分岐するために性能が低下するのを防ぐ。

【解決手段】投機命令および投機誤りをチェックするチェック命令を有するプロセッサ向けのコードを生成するコンパイラにおいて、（１）プログラム中の繰り返し実行される部分に対し、投機命令およびチェック命令を使用したコード(a)と、投機命令およびチェック命令を使用しないコード(b)を含む、少なくとも２種類のパターンのコードを生成する処理と、（２）コード(a)の実行中に投機誤りチェックにかかった回数が特定の条件を満たす場合に、以降の繰り返しではコード(b)を実行するように、制御の移行を行うコードを生成する処理を含む。

【選択図】　図７

出願人履歴情報

識別番号　〔０００００５１０８〕

１．変更年月日　１９９０年　８月３１日
［変更理由］　新規登録
住　所　東京都千代田区神田駿河台４丁目６番地
氏　名　株式会社日立製作所